Part I

Antivirus Basics

In This Part

Chapter 1: Introduction to Antivirus Software
Chapter 2: Reverse-Engineering the Core
Chapter 3: The Plug-ins System
Chapter 4: Understanding Antivirus Signatures
Chapter 5: The Update System
Antivirus software is designed to prevent computer infections by detecting malicious software, commonly called malware, on your computer and, when appropriate, removing the malware and disinfecting the computer. Malware, also referred to as samples in this book, can be classified into various kinds, namely, Trojans, viruses (infectors), rootkits, droppers, worms, and so on.

This chapter covers what antivirus (AV) software is and how it works. It offers a brief history of AV software and a short analysis of how it evolved over time.

What Is Antivirus Software?

Antivirus software is special security software that aims to give better protection than that offered by the underlying operating system (such as Windows or Mac OS X). In most cases, it is used as a preventive solution. However, when that fails, the AV software is used to disinfect the infected programs or to completely clean malicious software from the operating system.

AV software uses various techniques to identify malicious software, which often self-protects and hides deep in an operating system. Advanced malware may use undocumented operating system functionality and obscure techniques in order to persist and avoid being detected. Because of the large attack surface these days, AV software is designed to deal with all kinds of malicious payloads coming from both trusted and untrusted sources. Some malicious inputs that
AV software tries to protect an operating system from, with varying degrees of success, are network packets, email attachments, and exploits for browsers and document readers, as well as executable programs running on the operating system.

**Antivirus Software: Past and Present**

The earliest AV products were simply called *scanners* because they were command-line scanners that tried to identify malicious patterns in executable programs. AV software has changed a lot since then. For example, many AV products no longer include command-line scanners. Most AV products now use graphical user interface (GUI) scanners that check every single file that is created, modified, or accessed by the operating system or by user programs. They also install firewalls to detect malicious software that uses the network to infect computers, install browser add-ons to detect web-based exploits, isolate browsers for safe payment, create kernel drivers for AV self-protection or sandboxing, and so on.

Since the old days of Microsoft DOS and other antiquated operating systems, software products have evolved alongside the operating systems, as is natural. However, AV software has evolved at a remarkable rate since the old days because of the incredible amount of malware that has been created. During the 1990s, an AV company would receive only a handful of malware programs in the space of a week, and these were typically file infectors (or viruses). Now, an AV company will receive thousands of unique malicious files (unique considering their cryptographic hash, like MD5 or SHA-1) daily. This has forced the AV industry to focus on automatic detection and on creating *heuristics* for detection of as-yet-unknown malicious software by both dynamic and static means. Chapters 3 and 4 discuss how AV software works in more depth.

The rapid evolution of malware and anti-malware software products is driven by a very simple motivator: money. In the early days, virus creators (also called *virusers*) used to write a special kind of file infector that focused on performing functions not previously done by others in order to gain recognition or just as a personal challenge. Today, malware development is a highly profitable business used to extort money from computer users, as well as steal their credentials for various online services such as eBay, Amazon, and Google Mail, as well as banks and payment platforms (PayPal, for example); the common goal is to make as much money as possible.

Some players in the malware industry can steal email credentials for your Yahoo or Gmail accounts and use them to send spam or malicious software to thousands of users in your name. They can also use your stolen credit card information to issue payments to other bank accounts controlled by them or to pay *mules* to move the stolen money from dirty bank accounts to clean ones, so their criminal activity becomes harder to trace.
Another increasingly common type of malware is created by governments, shady organizations, or companies that sell malware (spying software) to governments, who in turn spy on their own people’s communications. Some software is designed to sabotage foreign countries’ infrastructures. For example, the notorious Stuxnet computer worm managed to sabotage Iran’s Natanz nuclear plant, using up to five zero-day exploits. Another example of sabotage is between countries and companies that are in direct competition with another company or country or countries, such as the cyberattack on Saudi Aramco, a sabotage campaign attributed to Iran that targeted the biggest oil company in Saudi Arabia.

Software can also be created simply to spy on government networks, corporations, or citizens; organizations like the National Security Agency (NSA) and Britain’s Government Communications Headquarters (GCHQ), as well as hackers from the Palestine Liberation Army (PLA), engage in these activities almost daily. Two examples of surveillance software are FinFisher and Hacking Team. Governments, as well as law enforcement and security agencies, have purchased commercial versions of FinFisher and Hacking Team to spy on criminals, suspects, and their own citizens. An example that comes to mind is the Bahrain government, which used FinFisher software to spy on rebels who were fighting against the government.

Big improvements and the large amounts of money invested in malware development have forced the AV industry to change and evolve dramatically over the last ten years. Unfortunately, the defensive side of information security, where AV software lies, is always behind the offensive side. Typically, an AV company cannot detect malware that is as yet unknown, especially if there is some quality assurance during the development of the malware software piece. The reason is very simple: AV evasion is a key part of malware development, and for attackers it is important that their malware stay undetected as long as possible. Many commercial malware packages, both legal and illegal, are sold with a window of support time. During that support period, the malware product is updated so it bypasses detection by AV software or by the operating system. Alternatively, malware may be updated to address and fix bugs, add new features, and so on. AV software can be the target of an attack, as in the case of The Mask, which was government-sponsored malware that used one of Kaspersky’s zero-day exploits.

**Antivirus Scanners, Kernels, and Products**

A typical computer user may view the AV software as a simple software suite, but an attacker must be able to view the AV on a deeper level.

This chapter will detail the various components of an AV, namely, the kernel, command-line scanner, GUI scanner, daemons or system services, file system filter drivers, network filter drivers, and any other support utility that ships with it.
ClamAV, the only open-source AV software, is an example of a scanner. It simply performs file scanning to discover malicious software patterns, and it prints a message for each detected file. ClamAV does not disinfect or use a true (behavioral-based) heuristic system.

A kernel, on the other hand, forms the core of an AV product. For example, the core of ClamAV is the libclam.so library. All the routines for unpacking executable programs, compressors, cryptors, protectors, and so on are in this library. All the code for opening compressed files to iterate through all the streams in a PDF file or to enumerate and analyze the clusters in one OLE2 container file (such as a Microsoft Word document) are also in this library. The kernel is used by the scanner clamscan, by the resident (or daemon) clamd, or by other programs and libraries such as its Python bindings, which are called PyClamd.

**NOTE** AV products often use more than one AV core or kernel. For example, F-Secure uses its own AV engine and the engine licensed from BitDefender.

An antivirus product may not always offer third-party developers direct access to its core; instead, it may offer access to command-line scanners. Other AV products may not give access to command-line scanners; instead, they may only allow access to the GUI scanner or to a GUI program to configure how the real-time protection, or another part of the product, handles malware detection and disinfection. The AV product suite may also ship with other security programs, such as browsers, browser toolbars, drivers for self-protection, firewalls, and so on.

As you can see, the product is the whole software package the AV company ships to the customer, while the scanners are the tools used to scan files and directories, and the kernel includes the core features offered to higher-level software components such as the GUI or command-line scanners.

### Typical Misconceptions about Antivirus Software

Most AV users believe that security products are bulletproof and that just installing AV software keeps their computers safe. This belief is not sound, and it is not uncommon to read comments in AV forums like, “I'm infected with XXX malware. How can it be? I have YYY AV product installed!”

To illustrate why AV software is not bulletproof, let’s take a look at the tasks performed by modern AV products:

- Discovering known malicious patterns and bad behaviors in programs
- Discovering known malicious patterns in documents and web pages
- Discovering known malicious patterns in network packets
- Trying to adapt and discover new bad behaviors or patterns based on experience with previously known ones
You may have noticed that the word known is used in each of these tasks. AV products are not bulletproof solutions to combat malware because an AV product cannot identify what is unknown. Marketing material from various AV products may lead the average users to think they are protected from everything, but this is unfortunately far from true. The AV industry is based on known malware patterns; an AV product cannot spot new unknown threats unless they are based on old known patterns (either behavioral or static), regardless of what the AV industry advertises.

**Antivirus Features**

All antivirus products share a set of common features, and so studying one system will help you understand another system. The following is a short list of common features found in AV products:

- The capability to scan compressed files and packed executables
- Tools for performing on-demand or real-time file or directory scanning
- A self-protection driver to guard against malware attacking the actual AV
- Firewall and network inspection functionality
- Command-line and graphical interface tools
- A daemon or service
- A management console

The following sections enumerate and briefly discuss some common features shared by most AV products, as well as more advanced features that are available only in some products.

**Basic Features**

An antivirus product should have some basic features and meet certain requirements in order to be useable. For example, a basic requirement is that the AV scanner and kernel should be fast and consume little memory.

**Making Use of Native Languages**

Most AV engines (except the old Malwarebytes software, which was not a full AV product) are written in non-managed/native languages such as C, C++, or a mix of both. AV engines must execute as quickly as possible without degrading the system’s performance. Native languages fulfill these requirements because, when code is compiled, they run natively on the host CPU at full speed. In the
case of managed software, the compiled code is emitted into a bytecode format and requires an extra layer to run: a virtual machine interpreter embedded in the AV kernel that knows how to execute the bytecode.

For example, Android DEX files, Java, and .NET-managed code all require some sort of virtual machine to run the compiled bytecode. This extra layer is what puts native languages ahead of managed languages. Writing code using native languages has its drawbacks, though. It is harder to code with, and it is easier to leak memory and system resources, cause memory corruption (buffer overflows, use-after-free, double-free), or introduce programming bugs that may have serious security implications. Neither C nor C++ offers any mechanism to protect from memory corruptions in the way that managed languages such as .NET, Python, and Lua do. Chapter 3 describes vulnerabilities in the parsers and reveals why this is the most common source of bugs in AV software.

**Scanners**

Another common feature of AV products is the scanner, which may be a GUI or command-line on-demand scanner. Such tools are used to scan whenever the user decides to check a set of files, directories, or the system’s memory. There are also on-access scanners, more typically called residents or real-time scanners. The resident analyzes files that are accessed, created, modified, or executed by the operating system or other programs (like web browsers); it does this to prevent the infection of document and program files by viruses or to prevent known malware files from executing.

The resident is one of the most interesting components to attack; for example, a bug in the parser of Microsoft Word documents can expose the resident to arbitrary code execution after a malicious Word document is downloaded (even if the user doesn’t open the file). A security bug in the AV’s email message parser code may also trigger malicious code execution when a new email with a malicious attachment arrives and the temporary files are created on disk and analyzed by the on-access scanner. When these bugs are triggered, they can be used as a denial-of-service attack, which makes the AV program crash or loop forever, thus disarming the antivirus temporarily or permanently until the user restarts it.

**Signatures**

The scanner of any AV product searches files or packets using a set of signatures to determine if the files or packets are malicious; it also assigns a name to a pattern. The signatures are the known patterns of malicious files. Some typical, rather basic, signatures are consumed by simple pattern-matching techniques (for example, finding a specific string, like the EICAR string), CRCs (checksums), or MD5 hashes. Relying on cryptographic hashes, like MD5, works for only a
specific file (as a cryptographic hash tries to identify just that file), while other fuzzy logic-based signatures, like when applying the CRC algorithm on specific chunks of data (as opposed to hashing the whole file), can identify various files.

AV products usually have different types of signatures, as described in Chapter 8. These signature types range from simple CRCs to rather complex heuristics patterns based on many features of the PE header, the complexity of the code at the entry point of the executable file, and the entropy of the whole file or some section or segment in the executable file. Sometimes signatures are also based on the basic blocks discovered while performing code analysis from the entry point of the executable files under analysis, and so on.

Each kind of signature has advantages and disadvantages. For example, some signatures are very specific and less likely to be prone to a false positive (when a healthy file is flagged as malware), while others are very risky and can generate a large list of false positives. Imagine, for example, a signature that finds the word Microsoft anywhere in a file that starts with the bytes MZ\x90. This would cause a large list of false positives, regardless of whether it was discovered in a malware file. Signatures must be created with great care to avoid false positives, like the one in Figure 1-1, or true negatives (when true malware code is flagged as benign).

![Figure 1-1: A false positive generated with Comodo Internet Security and the de facto reverse-engineering tool IDA](image)

**Compressors and Archives**

Another key part of every AV kernel is the support for compressed or archived file formats: ZIP, TGZ, 7z, XAR, and RAR, to name just a few. AVs must be able to decompress and navigate through all the files inside any compressed or archived file, as well as compressed streams in PDF files and other file formats. Because AV kernels must support so many different file formats, vulnerabilities are often found in the code that deals with this variety of input.

This book discusses various vulnerabilities that affect different AV products.
Unpackers

An unpacker is a routine or set of routines developed for unpacking protected or compressed executable files. Malware in the form of executables is commonly packed using freely available compressors and protectors or proprietary packers (obtained both legally and illegally). The number of packers an AV kernel must support is even larger than the number of compressors and archives, and it grows almost every month with the emergence of new packers used to hide the logic of new malware.

Some packer tools, such as UPX (the Universal Unpacker), just apply simple compression. Unpacking samples compressed by UPX is a very simple and straightforward matter. On the other hand, there are very complex pieces of software packers and protectors that transform the code to be packed into bytecode and then inject one or more randomly generated virtual machines into the executable so it runs the original code that the malware wrote. Getting rid of this virtualization layer and uncovering the logic of the malware is very hard and time-consuming.

Some packers can be unpacked using the CPU emulator of the AV kernel (a component that is discussed in the following sections); others are unpacked exclusively via static means. Other more complex ones can be unpacked using both techniques: using the emulator up to some specific layer and then using a static routine that is faster than using the emulator when some specific values are known (such as the size of the encrypted data, the algorithm used, the key, and so on).

As with compressors and archives, unpackers are a very common area to explore when you are looking for vulnerabilities in AV software. The list of packers to be supported is immense; some of them are used only during some specific malware campaign, so the code is likely written once and never again verified or audited. The list of packers to be supported grows every year.

Emulators

Most AV kernels on the market offer support for a number of emulators, with the only exception being ClamAV. The most common emulator in AV cores is the Intel x86 emulator. Some advanced AV products can offer support for AMD64 or ARM emulators. Emulators are not limited to regular CPUs, like Intel x86, AMD64, or ARM; there are also emulators for some virtual machines. For example, some emulators are aimed at inspecting Java bytecode, Android DEX bytecode, JavaScript, and even VBScript or Adobe ActionScript.

Fingerprinting or bypassing emulators and virtual machines used in AV products is an easy task: you just need to find some incongruities here and there. For example, for the Intel x86 emulator, it is unlikely, if not impossible, that the developers of the AV kernel would implement all of the instructions supported by to-be-emulated CPUs in the same way the manufacturers of those
specific CPUs do. For higher-level components that use the emulator, such as the execution environments for ELF or PE files, it is even less likely that the developers would implement the whole operating system environment or every API provided by the OS. Therefore, it is really easy to discover many different ways to fool emulators and to fingerprint them. Many techniques for evading AV emulators are discussed in this book, as are techniques for fingerprinting them. Part 3 of this book covers writing exploits for a specific AV engine.

**Miscellaneous File Formats**

Developing an AV kernel is very complex. The previous sections discussed some of the common features shared by AV cores, and you can imagine the time and effort required to support these features. However, it is even worse with an AV kernel; the kernel must support a very long list of file formats in order to catch exploits embedded in the files. Some file formats (excluding compressors and archives) that come to mind are OLE2 containers (Word or Excel documents); HTML pages, XML documents, and PDF files; CHM help files and old Microsoft Help file formats; PE, ELF, and MachO executables; JPG, PNG, GIF, TGA, and TIFF image file formats; ICO and CUR icon formats; MP3, MP4, AVI, ASF, and MOV video and audio file formats; and so on.

Every time an exploit appears for some new file format, an AV engineer must add some level of support for this file format. Some formats are so complex that even their original author may have problems correctly handling them; two examples are Microsoft and its Office file formats, and Adobe and its PDF format. So why would AV developers be expected to handle it better than the original author, considering that they probably have no previous knowledge about this file format and may need to do some reverse-engineering work? As you can guess, this is the most error-prone area in any AV software and will remain so for a long time.

**Advanced Features**

The following sections discuss some of the most common advanced features supported by AV products.

**Packet Filters and Firewalls**

From the end of the 1990s up until around 2010, it was very common to see a new type of malware, called *worms*, that abused one or more remote vulnerabilities in some targeted software products. Sometimes these worms simply used default username-and-password combinations to infect network shares in Windows CIFS networks by copying themselves with catchy names. Famous examples are “I love you,” Conficker, Melissa, Nimda, Slammer, and Code Red.
Because many worms used network resources to infect computers, the AV industry decided to inspect networks for incoming and outgoing traffic. To do so, AV software installed drivers for network traffic analysis, and firewalls for blocking and detecting the most common known attacks. As with the previously mentioned features, this is a good source of bugs, and today worms are almost gone. This is a feature in AV products that has not been updated in years; as a result, it is likely suffering from a number of vulnerabilities because it has been practically abandoned. This is one of the remotely exposed attack surfaces that are analyzed in Chapter 11.

**Self-Protection**

As AV software tries to protect computer users from malware, the malware also tries to protect itself from the AV software. In some cases, the malware will try to kill the processes of the installed AV product in order to disable it. Many AV products implement self-protection techniques in kernel drivers to prevent the most common killing operations, such as issuing a call to ZwTerminateProcess. Other self-protection techniques used by AV software can be based on denying calls to OpenProcess with certain parameters for their AV processes or preventing WriteProcessMemory calls, which are used to inject code in a foreign process.

These techniques are usually implemented with kernel drivers; the protection can also be implemented in userland. However, relying on code running in userland is a failing protection model that is known not to have worked since 2000; in any case, many AV products still make this mistake. Various AV products that experience this problem are discussed in Part III of this book.

**Anti-Exploiting**

Operating systems, including Windows, Mac OS X, and Linux, now offer anti-exploiting features, also referred to as security mitigations, like Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP), but this is a recent development. This is why some AV suites offer (or used to offer) anti-exploiting solutions. Some anti-exploiting techniques can be as simple as enforcing ASLR and DEP for every single program and library linked to the executable, while other techniques are more complex, like user- or kernel-land hooks to determine if some action is allowed for some specific process.

Unfortunately, as is common with AV software, most anti-exploiting toolkits offered by the AV industry are implemented in userland via function hooking; the Malwarebytes anti-exploiting toolkit is one example. With the advent of the Microsoft Enhanced Mitigation Experience Toolkit (EMET), most anti-exploiting toolkits implemented by the AV industry either are incomplete compared to it or are simply not up to date, making them easy to bypass.
In some cases, using anti-exploiting toolkits implemented by some AV companies is even worse than not using any anti-exploiting toolkit at all. One example is the Sophos Buffer Overflow Protection System (BOPS), an ASLR implementation. Tavis Ormandy, a prolific researcher working for Google, discovered that Sophos installed a system-wide Dynamic Link Library (DLL) without ASLR being enabled. This system-wide DLL was injected into processes in order to enforce and implement a faux ASLR for operating systems without ASLR, like Windows XP. Ironically, this system-wide DLL was itself compiled without ASLR support; as a result, in operating systems offering ASLR, like Windows Vista, ASLR was effectively disabled because this DLL was not ASLR enabled.

More problems with toolkit implementations in AV software are discussed in Part IV of this book.

Summary

This introductory chapter talked about the history of antiviruses, various types of malware, and the evolution of both the AV industry and the malware writers’ skills who seem to be always ahead of their game. In the second part of this chapter, the antivirus suite was dissected, and its various basic and advanced features were explained in an introductory manner, paving the way for more detailed explanation in the subsequent chapters of the book.

In summary:

- Back in the old days when the AV industry was in its infancy, the AVs were called scanners because they were made of command-line scanners and a signature database. As the malware evolved, so did the AV. AV software now includes heuristic engines and aims at protecting against browser exploits, network packets, email attachments, and document files.
- There are various types of malicious software, such as Trojans, malware, viruses, rootkits, worms, droppers, exploits, shellcode, and so on.
- Black hat malware writers are motivated by monetary gains and intellectual property theft, among other motivations.
- Governments also participate in writing malware in the form of spying or sabotage software. Often they write malware to protect their own interests, like the Bahrain government used the FinFisher software to spy on rebels or to sabotage other countries’ infrastructures as in the case of the Stuxnet malware that was allegedly co-written by the U.S. and the Israeli governments to target the Iranian nuclear program.
- AV products are well marketed using all sort of buzz words. This marketing strategy can be misleading and gives the average users a false sense of security.
- An AV software is a system made of the core or the kernel, which orchestrates the functionality between all the other components: plug-ins, system services, file system filter drivers, kernel AV components, and so on.

- AV need to run fast. Languages that compile into native code are the best choice because they compile natively on the platform without the overhead of interpreters (such as VM interpreters). Some parts of the AV can be written using managed or interpreted languages.

- An AV software is made up of basic features such as the core or the kernel, the scanning engine, signatures, decompressors, emulators, and support for various file format parsing. Additionally, AV products may offer some advanced features, such as packet inspection capabilities, browser security add-ons, self-protection, and anti-exploitation.

The next chapter starts discussing how to reverse-engineer AV cores kernels for the sake of automated security testing and fuzzing. Fuzzing is just one way to detect security bugs in antiviruses.